A model-based approach for developing event-driven architectures with AsyncAPI

Abel Gómez
agomezll@uoc.edu
Internet Interdisciplinary Institute (IN3)
Universitat Oberta de Catalunya (UOC)
Barcelona, Spain

Markel Iglesias-Urkia
miglesias@ikerlan.es
Ikerlan Technology Research Centre,
Basque Research and Technology Alliance (BRTA)
Arrasate-Mondragón, Spain

Aitor Urbieta
aurbieta@ikerlan.es
Ikerlan Technology Research Centre,
Basque Research and Technology Alliance (BRTA)
Arrasate-Mondragón, Spain

Jordi Cabot
jordi.cabot@icrea.cat
ICREA
Internet Interdisciplinary Institute (IN3)
Universitat Oberta de Catalunya (UOC)
Barcelona, Spain

ABSTRACT

In this Internet of Things (IoT) era, our everyday objects have evolved into the so-called cyber-physical systems (CPS). The use and deployment of CPS has especially penetrated the industry, giving rise to the Industry 4.0 or Industrial IoT (IIoT). Typically, architectures in IIoT environments are distributed and asynchronous, communication being guided by events such as the publication of (and corresponding subscription to) messages.

While these architectures have some clear advantages (such as scalability and flexibility), they also raise interoperability challenges among the agents in the network. Indeed, the knowledge about the message content and its categorization (topics) gets diluted, leading to consistency problems, potential losses of information and complex processing requirements on the subscriber side to try to understand the received messages.

In this paper, we present our proposal relying on *AsyncAPI* to automate the design and implementation of these architectures using model-based techniques for the generation of (part of) event-driven infrastructures. We have implemented our proposal as an open-source tool freely available online.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; **Source code generation**; **Publish-subscribe / event-based architectures**.

KEYWORDS

publish-subscribe, cyber-physical systems (CPS), event-driven architectures, AsyncAPI, Industrial Internet of Things (IIoT)

1 INTRODUCTION

The emergence of the Internet of Things (IoT) [12] has dramatically changed how physical objects are conceived in our society and industry. In the new era of the IoT, every object becomes a complex cyber-physical system (CPS) [22], where both the physical characteristics and the software that manages them are highly intertwined. Nowadays, many everyday objects are in fact CPSs, which increasingly use sensors and interfaces (APIs) to interact and exchange data with the cloud [31].

The ideas behind the IoT have been especially embraced by industry in the so-called Industrial IoT (IIoT) or Industry 4.0 [23]. It is in this Industry 4.0 scenario where CPSs become especially relevant, mainly in control and monitoring tasks [25].

IIoT has largely contributed to the growing interest in event-driven architectures [26], best used for asynchronous communication. Indeed, in order to achieve higher degrees of scalability, CPSs are typically deployed on event-driven asynchronous architectures that improve the overall behavior and reliability of systems. One of the most popular paradigms today is the so-called *publish-subscribe* [1] – followed by, for example, the *Message Queuing Telemetry Transport* (MQTT) protocol – where messages that are sent to and from a CPS are not directed to a certain recipient, but are proactively published and consumed by the agents involved according to certain criteria or categories. However, although these distributed architectures are especially scalable and tolerant to changes, they are not problem-free: since communication is done between equals, there must be an agreement between all parties on what are the expected message categories, as well as on their internal format and structure.

This is a key challenge we face at *Ikerlan*. As a technology center, *Ikerlan* currently coordinates different projects developing solutions to monitor, control and supervise systems of remote IoT devices in manufacturing plants, consumer goods, warehouses or smart buildings. Such solutions must support environments where a large number of devices send and consume data (e.g., sensor information, batch processed data, etc.). Following the current trend, the solutions developed are based on event-driven architectures following the *publish-subscribe* paradigm.

Based on our experience in working on a large number of IoT projects of different sizes and domains, in this article we present a model-based proposal to design and develop these architectures efficiently. Our solution relies on the *AsyncAPI Specification* [3] to formalize and (semi)automate the design and implementation of these architectures. Our proposal is the first model-based solution for the *AsyncAPI Specification*, and one of the first fully working

code generators for it as of 2020. We have implemented our proposal as an open-source tool freely available online.

This paper is structured as follows. Section 2 motivates this work by introducing a small use case that will be used as a running example throughout the rest of the paper; while Section 3 presents *AsyncAPI*, which serves as the background for this work. Section 4 presents our proposed workflow for designing and developing event-driven architectures; and Section 5 presents in detail our proposal, describing all the main artifacts and components involved. Section 6 presents the related work; Section 7 discusses on our experience and findings; and, finally, Section 8 concludes the paper.

2 MOTIVATION

Monitoring and control needs, as well as security and reliability requirements, make possible – and even desirable – to reuse a generic reference architecture in IIoT environments. Architectures in these environments are typically event-based, thus allowing a low coupling among the elements in the architecture. One common event-based paradigm is the so-called *publish-subscribe*, where messages are not directly sent to the recipients who will consume them, but are published under a certain category called *topic*. Only the devices subscribed to a certain *topic* will receive the messages published under it.

A common use case of these event-based architectures is IoT devices publishing monitoring and status data. Such data, whose volume can be very high, may be consumed by a cloud application that filters and processes it. On the other hand, control messages may be sent to reconfigure the IoT devices through a *Frontend* when needed. The central element of this architecture would be the *message broker*: the element in charge of managing publications, subscriptions, and the flow of messages between the elements of the network.

***Example**. To illustrate the architectures explained above, we will use a simplified use case of a factory with different production lines from an actual industrial partner of* Ikerlan *as shown in Figure 1. The* message broker *is shown in the center of the figure, and as aforementioned, it is in charge of managing the different publications and subscriptions. Different production lines, which contain a varying number of* presses*, are depicted on the right-hand side of the figure.* IoTBoxes *are IoT devices which are distributed throughout the factory, and are capable of monitoring and controlling different production lines. In the example,* IoTBox 1 *controls and monitors* Line A *(and thus presses* A1 *and* A2*), and* IoTBox 2 *controls and monitors* Line B *(and as a consequence, press* B1*). The* IoTBoxes *periodically collect data from the presses in the production line – e.g., pressure values and temperatures. These data are sent to be further processed in the* cloud*. To do this,* IoTBox 1 *publishes its monitoring information in topic* `iotbox/box1/monitor`*, while* IoTBox 2 *does the same on topic* `iotbox/box2/monitor`*. To receive the monitoring data, the* cloud *is subscribed to topic* `iotbox/+/monitor` *– the* `+` *symbol acts as a wildcard – and as a consequence, the* cloud *receives the monitoring data of both* IoTBoxes *under topics* `iotbox/box1/monitor` *and* `iotbox/box2/monitor`*.*

IoTBoxes *may also receive configuration commands – for example, to change the monitoring frequency. These configuration commands can be issued remotely via the* frontend*. For example, the frontend may publish the desired monitoring frequency for* Line A *– which is controlled by* IoTBox 1 *– by publishing a message with the correct format under topic* `iotbox/box1/config`*. Since* IoTBox 1 *is subscribed to this topic, it will receive the configuration command and will reconfigure itself as requested. The same applies to* IoTBox 2 *and* Line B*, but under topic* `iotbox/box2/config` *in this case.*

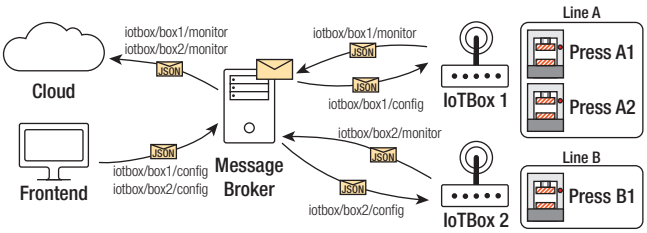

Figure 1: Example of an event-based architecture in IoT

Listing 1: Example message as to be published under `iotbox/box1/monitor`

```
{
 "id": "Line A",
 "presses": [
  {
   "id": "Press A1",
   "ts": "2020-05-11T18:01:06.158Z",
   "value": 10.0
  },{
   "id": "Press A2",
   "ts": "2020-05-11T18:01:06.329Z",
   "value": 15.5
  }
 ]
}
```

As it can be guessed by examining the example above, one of the major challenges that these architectures pose is **consistency** [17]: the format of the messages exchanged and the *topics* under which they are published and subscribed must be kept consistent throughout the life cycle of the system. Failure to comply with this could result in a system malfunction: if any of the agents introduces (even minimal) changes in the definition of communications, and these changes are not propagated to all the agents involved, interoperability problems inevitably occur. Communication can cease to be effective for two reasons: *(i)* because there is a **divergence in the *topics*** under which messages are published, thus resulting in agents not receiving messages they are interested in; or *(ii)* because there is a **divergence in the format** of the messages of a certain *topic*, and therefore these cannot be understood by the subscribers receiving them.

***Example**. Listing 1 shows a well-formed message as it is published by* IoTBox 1 *under topic* `iotbox/box1/monitor`*. The message contains a JSON object with two fields:* `id` *and* `presses`*.* `id` *identifies the production line, and* `presses` *is an array of objects, each one containing three fields: a press* `id`*, the timestamp (*`ts`*) when the measurement was taken on the press, and the* `value` *of the measured pressure in Pascals. An example of divergence in the topics is the case when the cloud subscribes to the wrong topic by mistake (e.g., to* `iotbox/box1/monitoring` *instead of* `iotbox/box1/monitor`*). In*

such case, the monitoring data of IoTBox 1 *will not be delivered to the cloud ever. An example of divergence in the format can happen if the developers coding the cloud application do not pay attention to the message format. For example, if the* `value` *is treated as an integer, the decimals will be lost; or if the timestamp is treated as a long with the epoch time (instead of as a formated string) a runtime error may occur.*

3 ASYNCAPI: TOWARDS A STANDARD LANGUAGE FOR DESCRIBING EVENT-BASED ARCHITECTURES

Aforementioned consistency issues are not unique to event-based architectures where communication occurs asynchronously. In fact, other architectures also manifest them, as it is the case of resource-oriented architectures where communication occurs synchronously. However, in these cases, the industry has already proposed standardized solutions to support the development of such architectures. An example is *OpenAPI* and its complete ecosystem. The *OpenAPI Specification* [29] is a description format for APIs based on the REST [13] paradigm that allows, among other things, to specify the operations offered by the API, the parameters of each operation, the authentication methods, etc.

For event-based architectures, and taking inspiration from *OpenAPI*, the *AsyncAPI Specification* [3] proposal has recently emerged as a promising alternative. *AsyncAPI* descriptions are expected to be both human and machine readable. To achieve this goal, files defining a message-driven API are represented as JSON objects and conform to the JSON standard[1]. Such files allow describing, among other things, the message brokers of an architecture, the *topics* of interest, or the different formats of the messages associated with each of the *topics*. Next, we introduce the most preeminent concepts – JSON objects in the *Specification* – of the *AsyncAPI* proposal for future reference:

The AsyncAPI object is the root document object of an API definition. It combines resource listing and API declaration together into one document. Its main fields are: *asyncapi*, to specify the *AsyncAPI Specification* version being used; *info*, an *Info* object; *servers*, a *Servers* object; *channels*, a *Channels* object; and *components*, a *Components* object.

The Info object provides the API metadata, such as its *title*, *version*, *description*, *termsOfService*, *contact*, and *license*.

The Servers object is a map of *Server* objects.

A Server object typically represents a message broker (or a similar computer program). This object is used to capture details such as URLs, protocols and security configuration of such brokers. Variable substitution is also supported. The object contains, among other fields, a *url* to the target host, its *protocol* (e.g., http, mqtt, stomp, kafka, etc.), the *protocolVersion*, a *description*, or a map of *variables*.

The Channels object is a map holding relative path names and individual *Channel Item* objects. *Channel* paths are relative to servers. *Channels* are also known as *topics*, *routing keys*, *event types* or *paths* depending on the protocol or technology used.

A Channel Item object describes the operations available on a single channel (i.e., *topic*). Typical fields are: *description*, to describe the channel; *subscribe*, an *Operation* object; *publish*, an Operation object too; or *parameters*, a map of the parameters included in the channel name.

An Operation object describes a publish or a subscribe operation. This provides a place to document how and why messages are sent and received. Most common fields are: *operationId*, a unique string used to identify the operation; *summary*, a short summary of what the operation is about; *description*, a verbose explanation of the operation; and message, a *Message* object with the definition of the message that will be published or received on this channel.

A Message object describes a message received on a given channel and operation. For a message, the following fields can be specified among others: *name*, a machine-friendly name for the message; *title*, a human-friendly title for the message; *summary*; *description*; or *payload*, which can be of any type but defaults to *Schema* object.

A Schema object allows the definition of input and output data types. These types can be objects, but also primitives and arrays. This object is a superset of the JSON Schema Specification Draft 7[2]. Typical fields of an *Schema* object are: *title*; *type* (any of "boolean", "integer", "number", "string", "object", "array" or "null"); *enum*, to limit possible values from a list of literals; *properties*, to specify the fields of objects; *maxItems* and *minItems*, to specify the cardinality of arrays; or items, to specify the schema of the array elements.

A Reference object is a simple object which allows referencing other components in the specification, internally and externally. It only contains the *$ref* field.

The Components object holds a set of reusable objects for different aspects of the AsyncAPI definition. Elements defined within the *Components* object can be referenced by using a *Reference* object. Reusable objects are mapped by name in their corresponding field. Some examples are: *schemas*, for *Schema* definitions; *messages*, for *Message* definitions; *parameters*, for *Parameters*; or *operationTraits* and *messageTraits*, which are traits that can be applied to operations and messages respectively, and are defined similarly to *Operations* and *Messages*.

Example. *Listing 2 shows, in a simplified way, how part of our running example is specified using AsyncAPI. To keep the example manageable, we have specified only the monitoring part (thus excluding the configuration topics – e.g.,* `iotbox/box1/config` *– and associated messages). As it can be seen, in line 2, we specify that the definition adheres to the* AsyncAPI Specification *version* `2.0.0`*, while in line 3, we specify the information of our API. As lines 4–6 specify, our infrastructure has a single server, called* `production` *– whose host name is* `example.com` *– with an MQTT broker listening on port* `1883`*. The rest of the* AsyncAPI *object specifies the topics exposed by our API, and the format of the messages that can be interchanged. This is done via the* `channels` *property (lines 7–24), which in turn, references some reusable artifacts that have been defined within the* `components`

[1] YAML, being a superset of JSON, can be used as well to represent an *AsyncAPI Specification* file too.

[2] https://json-schema.org/specification-links.html#draft-7

property (lines 25–65). Line 8 specifies the name of the only channel –i.e., topic– of our infrastructure: `iotbox/{id}/monitor`*. As it can be guessed,* `iotbox/{id}/monitor` *is a parameterized name, in which the* `{id}` *substring is substituted by the actual* IoTBox *name when publishing a message (thus publishing either under the* `iotbox/box1/monitor` *or* `iotbox/box2/monitor` *topics). Lines 10-13 specify the actual information of the parameter: in line 10, we specify its name; in line 11, we provide a description; and in line 12, we specify its type. The publish and subscribe operations are specified in lines 15–18 and 19–22 respectively. As both operations publish and receive the same kind of messages, they reference the reusable definition named* `statusMessage`*, which is defined in the* `messages` *property of the* `components` *object (lines 27–30). The payload of a* `statusMessage` *is a* `lineInfo` *object. The schema of* `lineInfo` *objects is specified in lines 33–46. A* `lineInfo` *is a JSON object with two properties:* `id`*, a string value; and* `presses`*, an array of* `pressInfo` *objects. As specified in lines 47–63, a* `pressInfo` *is an object with three properties: an* `id`*; a timestamp (whose name is* `ts`*); and a numeric value. As it can be observed, the example message shown in Section 2 – i.e., Listing 1 – contains a* `lineInfo` *object conforming to the specification in lines 33–63.*

Listing 2: AsyncAPI specification for an IoTbox

```
1  {
2   "asyncapi": "2.0.0",
3   "info": { "title": "IoTBox API", "version": "1.0.0" },
4   "servers": {
5    "production": { "protocol": "mqtt", "url": "example.com:1883" }
6   },
7   "channels": {
8    "iotbox/{id}/monitor": {
9     "parameters": {
10     "id": {
11      "description": "The ID of the IoTBox",
12      "schema": { "type": "string" }
13     }
14    },
15    "publish": {
16     "operationId": "publishStatus",
17     "message": { "$ref": "#/components/messages/statusMessage" }
18    },
19    "subscribe": {
20     "operationId": "subscribeStatus",
21     "message": { "$ref": "#/components/messages/statusMessage" }
22    }
23   }
24  },
25  "components": {
26   "messages": {
27    "statusMessage": {
28     "description": "Status of a given subsystem",
29     "payload": { "$ref": "#/components/schemas/lineInfo" }
30    }
31   },
32   "schemas": {
33    "lineInfo": {
34     "type": "object",
35     "properties": {
36      "id": {
37       "type": "string",
38       "description": "Identifier of the subsystem"
39      },
40      "presses": {
41       "type": "array",
42       "description": "Info of presses in this subsystem",
43       "items": { "$ref": "#/components/schemas/pressInfo" }
44      }
45     }
46    },
47    "pressInfo": {
48     "type": "object",
49     "properties": {
50      "id": {
51       "type": "string",
52       "description": "Identifier of the press"
53      },
54      "ts": {
55       "type": "string",
56       "title": "Timestamp"
57      },
58      "value": {
59       "type": "number",
60       "description": "Pressure of the press in Pascals"
61      }
62     }
63    }
64   }
65  }
66 }
```

As it can be observed in the example, the *AsyncAPI Specification* allows defining all the relevant information needed to design and execute an event-driven API. However, as of writing this manuscript, *AsyncAPI* is still in early stages of development, and therefore, its tooling is still underdeveloped: although some code generators exist, they are far from complete and most of the development effort has been put in the generation of documentation to be consumed by humans instead. This fact has hampered the impact of the *AsyncAPI* proposal in the design of a standardized development process for event-driven architectures.

4 ASYNCAPI AS THE *SINGLE SOURCE OF TRUTH* IN EVENT-DRIVEN ARCHITECTURES

One of the major flaws of event-driven architectures, as Section 2 illustrates, is how easily the knowledge about the infrastructure dilutes among all the elements involved in it. As a consequence, it is very easy to introduce divergences on how messages are sent and consumed by the different actors involved. This major flaw can, however, be solved if a *single source of truth* is used throughout the design, development and execution of the infrastructure. As we have seen in Section 3, despite its current limitations, *AsyncAPI* provides the grounds to design a complete conceptual framework that can be used as this *single source of truth*. Nevertheless, in order to make an effective use of *AsyncAPI*, a proper process and infrastructure supporting not only the design, but also the whole lifecycle of an event-driven architecture, is still needed. In order to fill this gap, we complement the *AsyncAPI* proposal with a model-based approach that allows us to overcome its current limitations. Based on our experience, using model-based techniques allows us *(i)* to keep our approach modular and extensible, alleviating a possible *vendor lock-in*; *(ii)* to integrate other IoT standards, different programming languages and frameworks, or future projects with not yet known requirements at *Ikerlan*; and *(iii)* to boost our productivity by taking advantage of the plethora of model-based technologies and solutions available in the market – metamodeling frameworks, code generation engines, model transformation engines, etc.

Figure 2 shows how we envision a development workflow where **the *AsyncAPI* definition of an event-driven architectures can**

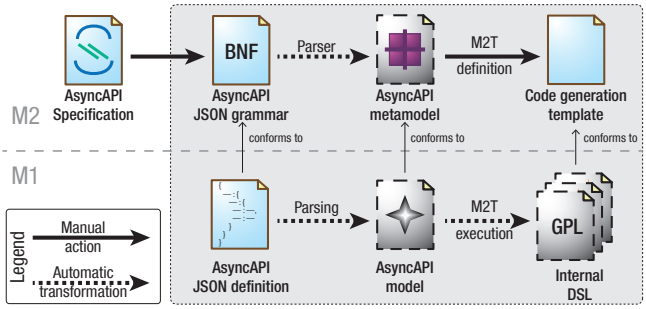

Figure 2: Development process and main involved artifacts

now be the only truly *single source of truth* that is shared among all the agents involved. The main involved artifacts in this development workflow are represented as a paper sheet shape. We call the tool implementing this workflow *AsyncAPI Toolkit*.

On the one hand, artifacts depicted in the figure with a solid line represent manually created artifacts, while artifacts depicted with a dashed line are automatically generated using different transformations. On the other hand, artifacts at the M1 layer (the model layer), are defined – or automatically created – each time a new development process is enacted; while artifacts at the M2 layer (the metamodel layer) are defined only once during the development of the toolkit itself.

Based on the *AsyncAPI Specification*, we propose to develop an *AsyncAPI JSON grammar* that validates event-driven architecture definitions conforming to the JSON-based *AsyncAPI Specification*. We will call this textual representation the *concrete syntax*. Likewise, from the *AsyncAPI Specification* an *AsyncAPI metamodel* can be developed so that we can take advantage of all the tooling the model-driven techniques provide us. This *AsyncAPI metamodel* will provide the *abstract syntax* for *AsyncAPI*. Using the concepts of this abstract syntax, a model-to-text (M2T) transformation generating executable code in a *general purpose language* (GPL) can be defined.

The executable code is a library managing all basic functionality of a concrete event-driven infrastructure. This library can be shared and reused by all the elements participating and collaborating in the architecture, and can be implemented in such a way that it exposes an internal *domain specific language* (DSL) that can be easily employed by the *client code*[3] to perform tasks such as message creation, message parsing and processing, publications, subscriptions, etc. For this latter step, we can again exploit the entire set of tools available in a modeling ecosystem to create code in any GPL.

It is noteworthy to remark that we have chosen a JSON-based concrete syntax for two main reasons: *(i)* we want to keep our solution fully compliant with the *AsyncAPI Specification*; and *(ii)* we consider that a *lightweight* modeling language will be better accepted by existing developers. This, however, does not hinder us from designing alternative richer concrete syntaxes – e.g., graphical modeling languages – that take advantage of the rest of our infrastructure (such as the *AsyncAPI metamodel* and code generators).

[3]We understand as *client code* the actual applications making use of the messages sent and received, and whose logic cannot be captured in the *AsyncAPI* definition. An example of client code would be the application running in the *cloud* which is in charge of processing the monitoring data sent by the *IoTBoxes*.

Example. *An architect willing to use AsyncAPI as the* single source of truth *in our factory use case would proceed as follows. The architect would use the JSON-based representation for* AsyncAPI *to define the event-driven architecture. If we consider only the monitoring part of our use case, in practice, this definition is exactly the one we show in Listing 2 since we propose to fully comply with the* AsyncAPI Specification. *While the user is editing, our* AsyncAPI Toolkit *creates the corresponding* AsyncAPI model *and executes the M2T transformation generating the internal DSL on-the-fly. The generated DSL – which is an executable library in a GPL such as Java – can be directly distributed in source code form or as packaged binaries to the developers of the different components of the architecture (e.g., the* IoTBoxes*, the cloud, or the frontend). Thus, a developer wanting to publish a message or consuming a message, does not need to care about other elements in the architecture or external documentation: all he or she has to do is to import the libraries of the DSL. The DSL will provide all the functionality needed to connect to a specific* broker*, create a specific message in the right format, and publish it, or vice-versa: connect to the specified* broker*, subscribe to a specific topic, and receive the messages in the right and native format of the platform being used.*

5 THE *ASYNCAPI TOOLKIT* UNDER THE MICROSCOPE

We have implemented the *AsyncAPI Toolkit* workflow as an open source solution[4]. As aforementioned, we have also followed model-driven development principles to create it: instead of manually developing a set of editors supporting the textual *AsyncAPI JSON Grammar* or the *AsyncAPI metamodel*, we have chosen the *Xtext framework* [15] to provide both a concrete and an abstract syntax for *AsyncAPI*. From an *AsyncAPI JSON Grammar*, *Xtext* can generate all the tooling (editor with content assist, parser, etc.) to easily create *AsyncAPI JSON definitions* that are automatically and transparently transformed to *AsyncAPI models* which conform to the *AsyncAPI metamodel*. The use of Xtext allows us to greatly reduce the development time to obtain a working textual editor capable of parsing *AsyncAPI Specifications*. Since Xtext uses the Eclipse Modeling Framework (EMF) [14], we can take advantage of all its ecosystem to develop the M2T transformations generating the internal DSL. Next, we describe in detail how our *AsyncAPI Toolkit* has been built[5] and how architects and developers can take advantage of it.

5.1 A JSON-based concrete syntax for AsyncAPI

The main manual step that must be done in Xtext to provide a concrete syntax for a textual language is the development of an Xtext grammar. Listing 3 shows an excerpt of the grammar[6] we have defined to support the definition of AsyncAPI in JSON following the concrete syntax proposed in [3].

[4]https://hdl.handle.net/20.500.12004/1/A/ASYNCAPI/001
[5]Only short illustrative excerpts of the different elements of the *AsyncAPI Toolkit* will be shown for brevity purposes. For a full reference, please check our repository.
[6]See https://www.eclipse.org/Xtext/documentation/301_grammarlanguage.html for a full reference of Xtext grammars.

Listing 3: Excerpt of the Xtext grammar

```
1  AsyncAPI:     // Rule name
2    {AsyncAPI}  // Output type name
3    '{'         // The curly brace character
4    (
5        ( '"asyncapi"' ':' version=VersionNumber ','? )
6      & ( '"info"' ':' info=Info ','? )
7      & ( '"servers"' ':' '{' servers+=Server (',' servers+=Server)*
             '}' ','? )?
8      & ( '"channels"' ':' '{' channels+=Channel (',' channels+=
           Channel)* '}' ','? )?
9      & ( '"components"' ':' components=Components ','? )?
10     // More properties omitted for brevity purposes...
11   )
12   '}';
13
14 // Info rule omitted for brevity purposes...
15
16 Server:
17   {Server} name=STRING ':' '{' (
18       ( '"url"' ':' url=STRING ','? )
19     & ( '"protocol"' ':' protocol=Protocol ','? )
20     & ( '"description"' ':' description=STRING ',' )?
21     // More properties omitted for brevity purposes...
22   ) '}';
23
24 // Channel and Components rule omitted for brevity purposes
25
26 enum VersionNumber:
27   _200 = '"2.0.0"';
28
29 enum Protocol:
30   amqp = '"amqp"'     | amqps = '"amqps"' | mqtt = '"mqtt"'
31   | mqtts = '"mqtts"' | ws = '"ws"'       | wss = '"wss"'
32   | stomp = '"stomp"' | stomps = '"stomps"';
33
34 terminal STRING:
35     '"' ( '\\' . | !('\\'|'"') )* '"'
36   | "'" ( '\\' . | !('\\'|"'") )* "'";
37
38 // Other rules omitted for brevity purposes...
```

In short, we have defined an Xtext rule for each one of the concepts defined in the *Schema* section of the aforementioned document. Listing 3 shows – in a simplified way – the rules to define in JSON an AsyncAPI specification version 2, with its *Info*, a set of *Servers*, *Channels*, and the *Components* Section. Taking as an example the *AsyncAPI* rule (line 1), in line 2 we specify that the application of the rule will produce an *AsyncAPI* object when parsing an input text, while lines 3 and 12 specify that a *AsyncAPI* is a textual element enclosed between the characters `{` and `}`. The parentheses in lines 4 and 11 denote an unordered group, i.e., the patterns between them, which are separated by an `&` symbol, may match only once and in any possible order. Line 5, for example, specifies that the version of an *AsyncAPI* is a sequence of characters starting with the `"asyncapi"` keyword, followed by a `:` symbol and followed by a text matching the *VersionInfo* rule (which in turn, is an *enumerated*, specifying that only version `"2.0.0"` is supported). The value of the parsed version number will be stored in an attribute of the *AsyncAPI* object named *version* of type *VersionNumber*. It is necessary to clarify some details of the grammar: first, to get advantage of the features provided by the Xtext unordered groups, we have defined the commas between groups as optional (`','?` expression near the end of each group); and second, we have relaxed the requirements of some elements marking them as optional (`?` symbol at the end of each group) to minimize the number of errors reported while parsing the input files for not overwhelming the users of the tool. These optionalities can be, however, later enforced programmatically so that the tool only accepts valid JSON instances.

Example. *As it can be seen, the* AsyncAPI *definition included in Listing 2 can be parsed by applying the rules of the grammar showed in Listing 3: the curly brace in line 1 of Listing 2 matches the curly brace in line 3 of Listing 3; the tokens in line 2 of Listing 2 (*`"asyncapi": "2.0.0"`*) match the tokens specified in line 5 of Listing 3; etc.*

5.2 An abstract syntax for AsyncAPI

Based on the grammar above, Xtext is able to generate an equivalent EMF-based metamodel. Fig. 3 shows the Ecore metamodel generated as a result of the *AsyncAPI JSON grammar*. As aforementioned, instances conforming to this metamodel are automatically created out of textual descriptions thanks to the tooling generated by Xtext. As it can be observed, it basically contains a class for each one of the rules defined in the Xtext grammar, and each one of the classes, contains the attributes specified in its corresponding rule. For example, among other elements, Fig. 3 shows the *AsyncAPI* and *Server* classes which correspond to the rules included in Listing 3, as well as the enumerations for *VersionInfo* and *Protocol*. The existence of this automatically generated metamodel enables the use of other EMF-based tools, such as model transformation engines.

It is noteworthy to mention that we have been extremely careful when defining the *AsyncAPI JSON grammar* so that the generated EMF-based metamodel closely represents the domain – i.e., the *AsyncAPI Specification* – and not only its textual representation. Thus, **this metamodel is not only an utility artifact that enables us to parse *AsyncAPI JSON specifications*, but it is a meaningful metamodel close to the concepts of the *AsyncAPI Specification***: as it can be seen, there is a direct match between it and the main concepts explained in Section 3, and as a consequence, we will not describe in detail the concepts shown in the figure to avoid being redundant. Having this meaningful metamodel allows us to build alternative concrete syntaxes that can reuse the existing tooling minimising our development effort.

Example. *The result of parsing the* AsyncAPI *definition in Listing 2 is partly shown in Figure 4. As it can be seen, the automatically generated model contains a single root* AsyncAPI *object, containing* Info, Server, Channel *and* Component *objects. The properties of the* `production` Server *are shown in the* Properties *tab. As the* AsyncAPI *definition in Listing 2 specifies, the name of the* Channel *object is* `iotbox/{id}/monitor`*, and it contains the operations* `publishOperation` *and* `subscribeOperation`*, and the* Parameter `id`*. The* Components *object contains the* `lineInfo` *and* `pressInfo` Schemas*, and the* `statusMessage` Message.

5.3 An internal Java DSL for effective event-driven communication

The last step of our proposed workflow is the generation of a library implementing an internal DSL which can be used and shared among all the elements in the infrastructure, and that will ensure that all participants know the same topics, and are able to create and

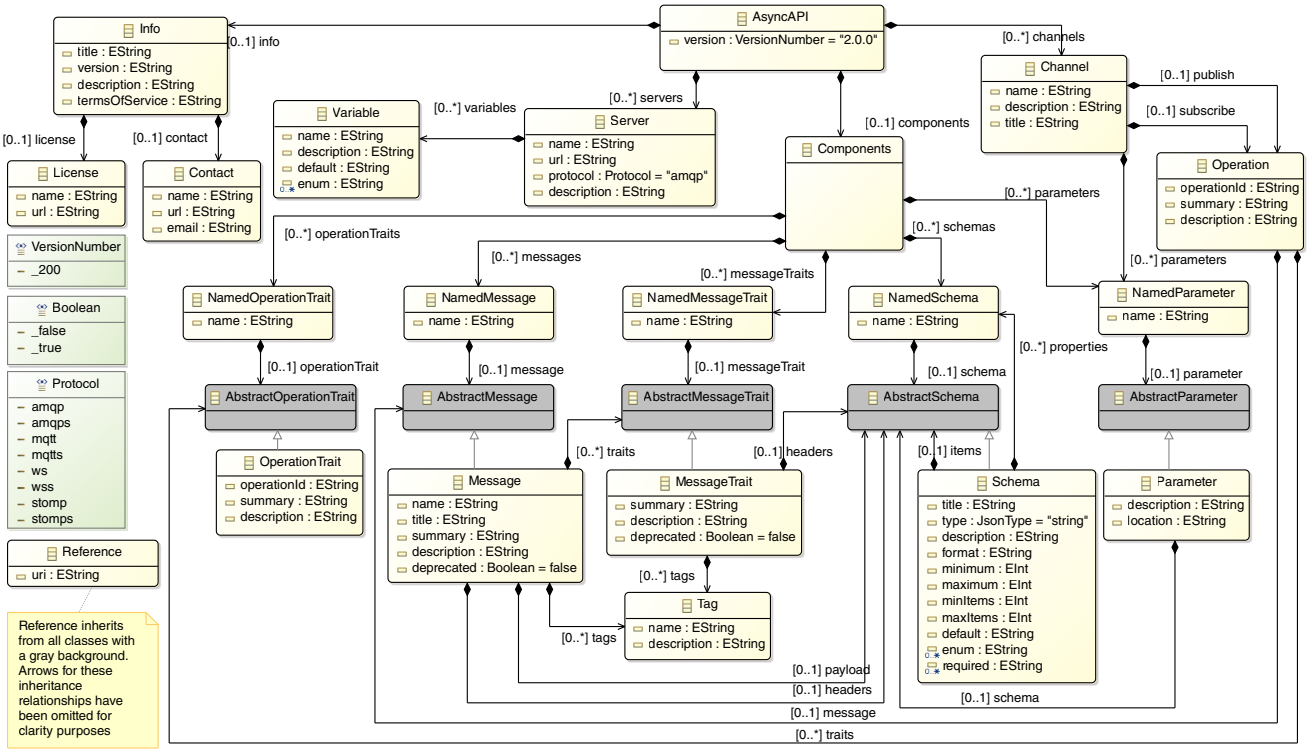

Figure 3: AsyncAPI metamodel

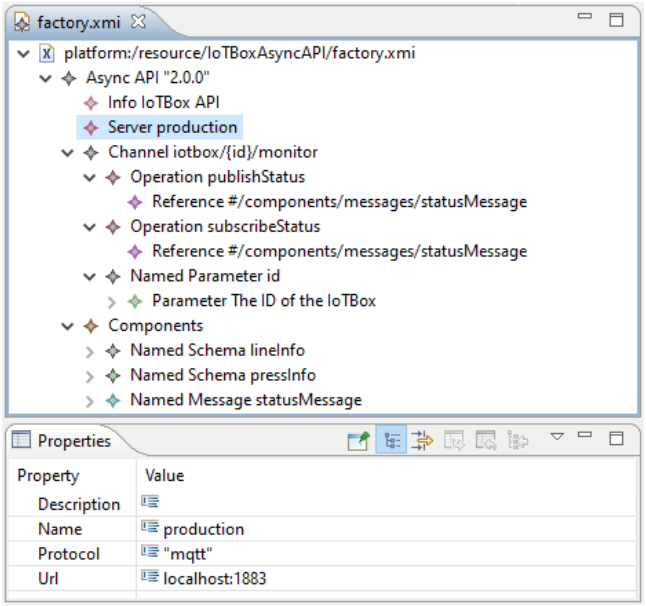

Figure 4: *AsyncAPI* definition of the factory use case represented as an instance of the *AsyncAPI metamodel*

consume messages conforming the same schemas and formats. Currently, our *AsyncAPI Toolkit* only supports the generation of Java code, but other languages and technologies can be easily plugged in by providing the proper code generation templates.

The M2T transformation applies the following rules:

A Server is chosen as the default *Server* of the architecture.

Channels are transformed to packages whose name matches the *Channel* – or topic – name. Hierarchies of topics – i.e., using slashes (/) – are respected.

Operations in *Channels* are transformed to classes inside the packages of their corresponding *Channels*. Classes for channel *Operations*, among other things, provide static operations for publishing and subscribing in their respective topics using the default *Server*. If *Channel* permits the use of *Parameters*, a nested class is generated to manage the parameter substitution and recovery.

Messages do not generate a Java artifact: since there is a direct correspondence between *Messages* and *Schemas* (via the *payload* association) *Messages* information is combined within the corresponding *Schema* when needed.

The Components object is transformed to a `components` package.

Schemas are transformed to immutable classes: they are cloneable and serializable (via Gson[7]), have getters but no setters, and have a private constructor. Instances are created via a dedicated nested *builder* class.

[7] https://github.com/google/gson

Schema classes can be generated directly in the `components` package if they are reusable, or nested. For example, an *Operation* can have nested declarations for its corresponding *Message* and *Schema*. In that case, the *Schema* class is declared as a nested class within the *Operation* class.

As it can be seen, the library provides specific artifacts for the most important concepts of the *AsyncAPI definition*: *operations* on each *channel – topic –* are unequivocally grouped in their specific packages; *payloads* of messages can only be created by using specific *builders*; and *messages* can only be sent and received using dedicated methods of specific classes.

Forcing developers to use builder classes and *getters* of the generated *Schema* classes – among other things – guarantees that the structure and the type of the messages is preserved and shared among all the elements of the architecture. The same applies, for example, with respect to the classes for publishing and subscribing since the relationship between *Channels*, *Operations*, *Messages* and *Schemas* is explicitly encoded in the internal DSL: static typing in Java ensures that developers do not mix incompatible types, or publish (or receive) messages in the wrong topics[8]. This approach seeks that developers keep their code up to date with respect to the latest version of the *AsyncAPI* definition used in the shared architecture.

***Example**. Listings 4 and 5 show two examples of client code using the internal DSL generated for our example* AsyncAPI *definition in Java (Listing 2), and more specifically, Listing 4 shows the code needed to create and publish the example message shown in Listing 1. As it can be seen in the imports (lines 5–6), there exists a* `iotbox._id_.monitor` *package (generated from the* `iotbox/{id}/monitor` Channel*), with a* `PublishStatus` *class (generated from the* `publishStatus` Operation*). Since the* `iotbox/{id}/monitor` *Channel has an* `id` *parameter, also a* `PublishStatusParams` *is created for the substitution and recovery of parameter values. It can also be seen that reusable* Schemas *(*`lineInfo` *and* `pressInfo`*) produce the corresponding classes in the* `schemas` *package (see lines 7–8).*

Thus, in order to create the message, developers only need to use the provided classes. For example, to create an instance of `LineInfo`*, a new* `LineInfoBuilder` *can be obtained by invoking* `LineInfo.newBuilder()` *(line 13), and then, it can be initialized by using the provided* fluent interface *[16] (e.g., methods* `withId` *and* `addToPresses`*). If any of the methods needs another object as an argument – such as for* `addToPresses` *– it can be created by using the corresponding builder as shown in lines 15–21. When a friendly name is available for a given* property *– for example, because a* title *was specified – the method provided by the fluent interface will use it instead of the actual Schema property name. Lines 18 and 25 are an example of this: timestamps are a property called* `ts`*, but the provided method is* `withTimestamp` *rather than* `withTs`*, thus making the code more understandable. Once all the properties have been set, the* `build()` *method is invoked. It is noteworthy to mention that validation logic – such as checking of required properties – could also be added in the* `build()` *method following a* fail fast *approach.*

Finally, once the payload of the message and the parameters have been created – parameters are created using a similar fluent interface *as shown in lines 30–31 – the* `publish` *operation can be invoked (see line 32). The* `publish` *method of the* `PublishStatus` *class only accepts instances of* `LineInfo` *as the first argument and instances of* `PublishStatusParams` *as the second argument. The publish operation will be in charge of doing the parameter substitution, and publishing the payload passed as an argument in the* `iotbox/box1/monitor`

[8] In any case, these checks can be proactively done in the generated code following a *fail fast* approach in the case these features are not natively provided by the language of the generated code (e.g., dynamically typed languages).

Listing 4: Main class of the *IoTBox* Publisher

```
1  package main;
2
3  import java.text.MessageFormat;
4  import java.time.Instant;
5  import iotbox._id_.monitor.PublishStatus;
6  import iotbox._id_.monitor.PublishStatus.PublishStatusParams;
7  import schemas.LineInfo;
8  import schemas.PressInfo;
9
10 public class Publish {
11   public static void main(String[] args) throws Exception {
12     LineInfo payload =
13       LineInfo.newBuilder()
14         .withId("Line A")
15         .addToPresses(
16           PressInfo.newBuilder()
17             .withId("Press A1")
18             .withTimestamp(Instant.now().toString())
19             .withValue(10.0)
20           .build()
21         )
22         .addToPresses(
23           PressInfo.newBuilder()
24             .withId("Press A2")
25             .withTimestamp(Instant.now().toString())
26             .withValue(15.5)
27           .build()
28         )
29       .build();
30     PublishStatusParams params =
31           PublishStatusParams.create().withId("box1");
32     PublishStatus.publish(payload, params);
33   }
34 }
```

Listing 5: Main class of the *IoTBox* Subscriber

```
1  package main;
2
3  import java.text.MessageFormat;
4  import iotbox._id_.monitor.SubscribeStatus;
5
6  public class Subscribe {
7    public static void main(String[] args) throws Exception {
8      SubscribeStatus.subscribe((params, message) -> {
9        System.out.println(MessageFormat.format("Message received
             from IoTBox ''{0}''!", params.getId()));
10       System.out.println(MessageFormat.format("Info about
             production line ''{0}'':", message.getId()));
11       message.getPresses().stream().forEach(
12         press -> System.out.println(
13           MessageFormat.format("At {0}, press ''{0}'' was pressing
                 at {2} Pa",
14             press.getTimestamp(),
15             press.getId(),
16             press.getValue()
17           )
18         )
19       );
20     });
21   }
22 }
```

topic. This will ensure that both the payload and the topic names will be syntactically correct and will match.

Listing 5 shows how an example application – such as one running in the cloud – will subscribe to the `iotbox/+/monitor` *topic and will receive messages sent to it. As it can be seen, it only needs to invoke the* `SubscribeStatus.subscribe` *method passing a callback function (expressed as a lambda expression in lines 8–20): the callback function will receive the value of the parameters in the* `params` *argument – which is of type* `SubscribeStatusParams` *– and the message payload in the* `message` *argument – which is of type* `LineInfo`*. From this point on, client code can make use of the getters provided by the generated code to retrieve all the information from them. As it can be seen, the example code only prints all the received information by the standard output.*

6 RELATED WORK

We compare our proposal with other works around API specifications, IIoT languages and code-generators and model-based approaches for communication.

As we will see, most of them focus on synchronous architectures while support for event-driven ones, like we propose, is much more limited.

6.1 REST APIs

AsyncAPI was inspired by the previous work on *OpenAPI* (formerly *Swagger*) [33]. *OpenAPI* allows to describe RESTful APIs [34] and includes several tools to assist developers, e.g., an editor, document generator, code generator, etc. proposed by the consortium itself and by a growing ecosystem of third-party providers (e.g., APIs.guru [2]). We also start to see model-based tools for *OpenAPI* [11] or [10].

However, for event-driven APIs, which are the ones that *AsyncAPI* addresses, such rich ecosystem does not exist yet.

6.2 Domain-Specific Languages for IIoT

As we propose, MDE has been used to accelerate the development process of industrial systems in the Industry 4.0 context. Among other works, the benefits of MDE for IIoT have been previously analyzed by Capilla et al. [4], and Young et al. [37], including some guidelines when modeling such systems to maximize their effectiveness [7].

In particular, several DSLs to model specific parts of IoT communication systems have been explored. One of such approaches is the one presented by Sneps-Sneppe and Namiot [32], where the authors present an extension of *Java Server Pages* to generate a web-based DSL to use in IoT applications. The proposed DSL enables IoT communications between the devices that support the process and the sensors. Negash et al. [27] also propose a DSL that is specifically designed for IoT, namely *DoS-IL*. However, they go further and also created an interpreter for the DOM, allowing the browser to be manipulated through scripts that interact with the DOM. *CREST*, presented by Klikovits et al. [24], is another DSL that aims to model CPSs of small scale that has synchronous evolution and reactive behavior.

However, these works focus on the data model, while the base for our approach are the messages and operations themselves. One exception is the work of Ivanchikj and Pautasso [21]. In their work, the authors present *RESTalk*, a DSL for modeling and visualizing RESTful conversations, i.e., a model of all possible sequences of HTTP message exchanges between client and servers. As before, RESTalk is based on the model of the *OpenAPI Specification*, and provides a visual and textual DSL. There are no message-oriented DSL solutions for event-driven architectures as the one we propose herein.

6.3 Automatic code generation for IIoT

Beyond modeling IIoT systems, some approaches go one step further and also target the partial code generation of the systems from such models.

This is the case of Ciccozzi and Spalazzese [5], where the authors present the *MDE4IoT* framework. As seen with others before, this framework focuses on the data model and not on the messages that need to be exchanged while the system is running. Another approach is *TRILATERAL* [17, 20], a tool that uses MDE with IoT communication protocols to generate artifacts for industrial CPSs. This tool allows using a visual editor to input a model based on the IEC 61850 standard for electrical substations and the tool automatically generates the C++ code that enables the devices to communicate using HTTP-REST, CoAP, or WS-SOAP (all of them synchronous). The Web of Things (WoT) [36] is an approach that tries to enable interoperability among devices sharing the definition of a common data model. It is still a work in progress, but there are some works that use it, such as Delicato et al. [8] and Iglesias-Urkia et al. [18, 19]. Nevertheless, the central building block in the WoT is the *Thing Description*, which defines the data model and the communication, but it is not targeted to event-driven architectures.

6.4 Model-based approaches for message-based architectures

Only a few works propose the use of MDE to describe the actual communication in an IIoT architecture, and not just the data. One of such examples is the one from Riedel et al. [30], where the authors present a tool that generates C, C# and Java code that uses SOAP Web Services (WS-SOAP) as the communication protocol. They propose the use of the *Essential Meta-Object Facility* (EMOF) [28] for data metamodels and EMF to generate the messages between IoT subsystems. The tool allows configuring which part of the generated code runs on the gateway and which part on the IoT node. However, although this approach is similar to ours in the sense that it also models the messages to exchange, the main difference relies in that theirs uses SOAP communication, which makes it synchronous communication instead of an asynchronous event-driven one. Another example is the work of Thramboulidis and Christoulakis [35], which integrates CPSs and IoT with a framework named *UML4IoT* that allows automating the process of generating CPSs. To do that, the CPSs are modeled using SysML and implemented using an object-oriented API that is later transformed to a RESTful API, using LWM2M for the communication, which is also synchronous.

To sum up, while there are a few works focusing on a goal similar to ours, they mostly target REST APIs or synchronous environments. Although there is some previous research on modeling

event-driven architectures, such as the one by Clark and Barn [6], to the best of our knowledge, ours is the first approach that is based on an event-driven approach – in this case using the *AsyncAPI Specification* – and that generates the necessary code automatically.

7 DISCUSSION

AsyncAPI Toolkit has been developed in the context of the *MegaM@Rt2*[9] project and has already been applied to some of its use cases and other internal projects at *Ikerlan*. This section discusses some of the benefits we have observed so far.

Lower development and deployment time. Adopting *AsyncAPI Toolkit* in a project significantly decreases the time to develop and deploy the software system. On the one hand, reusing the generic metamodel simplifies the definition of the schemas of the project and, with the automated code generation, there is no need to implement the boilerplate code in the client side, reducing manual tasks. Our preliminary observations[10] show a reduction in the development time of the infrastructure code on nearly a 66%, reducing the required development time from a few hours to less than an hour.

Obviously, there is are initial cost in developing *AsyncAPI Toolkit* itself, training the people in using it, and adapting the continuous integration infrastructure so that all components share the same version of the library implementing the internal DSL. But this cost is quickly compensated when using it over several projects, as it happens with any new MDE infrastructure [9].

Increasing code quality. In addition to lowering development time, the automated code generation usually is better structured and allows to reuse common code blocks, which increases code quality. The reused parts are already tested, hence, the validation process of the system is more simple. In addition, the generated code is also easier to maintain, as bugs or improvements in common parts need to be addressed only once. This leads to a reduction on engineering and maintenance costs.

In this sense, it has been detected that using the *AsyncAPI Toolkit* in different projects, the time to detect bugs has been decreased as more bugs are detected in the first stages of implementation and execution of the developed systems.

DSL benefits for Industry 4.0. Another upside is that, as diverse existing application domains share similarities – especially regarding communication requirements – the same solution can be applicable to all of them. In this context, we can more easily port our *AsyncAPI*-based solutions to a variety of related domains. This is especially interesting in the context of software product lines (SPLs).

Easy documentation. *AsyncAPI* includes a tool to automatically generate documentation out of an API. With it, given the specific model created by the toolkit (in JSON or YAML format) as input, *AsyncAPI* is able to generate its corresponding HTML or Markdown documentation.

At *Ikerlan* this feature has been regarded as an important aspect in the decision to move forward with the adoption of *AsyncAPI Toolkit*. This documentation capability together with the use of *AsyncAPI* as *single source of truth* enables all project participants (stakeholders, designers, architects, etc.) to share a common definition of the API, favouring interoperability and reducing the number of errors in the software development life cycle.

Requirement definition, validation and maintainability. Being an message-driven architecture, capturing the requirements can be directly done with the *AsyncAPI definition*. Therefore, there is no need to maintain a separate text document that needs to be interpreted by developers, as the requirements are specified in the *AsyncAPI definition* itself.

This has been another key reason to select *AsyncAPI* for ongoing and future projects at *Ikerlan*. Combined with the previous point, we see *AsyncAPI Toolkit* as a toolkit able to support most phases of the development cycle, from requirements to code-generation to – in the future – testing.

8 CONCLUSIONS

This article presents *AsyncAPI Toolkit*, a toolkit that allows specifying an event-driven API using *AsyncAPI* and generates code automatically. *AsyncAPI Toolkit* decreases the development time for projects that have monitoring and control requirements with asynchronous communication.

As future work, we plan to improve the usability of the process by having a complete Ecore-based importer that facilitates bootstrapping the *AsyncAPI definition* from existing domain models. Similarly, we will provide additional code generation templates to cover other languages beyond the current Java target. Besides code-generation, our *AsyncAPI metamodel* could be used to generate other software artifacts like test suites. Finally, we will perform additional empirical validations that help us better understand the specific trade-offs of introducing our model-based *AsyncAPI* infrastructure in new industrial projects.

ACKNOWLEDGMENTS

This project has received funding from the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No. 737494 – this Joint Undertaking receives support from the European Union's Horizon 2020 research and innovation program and from Sweden, France, Spain, Italy, Finland & Czech Republic – and from the Spanish government under project *Open Data for All* (RETOS TIN2016-75944-R).

REFERENCES

[1] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. 2015. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Communications Surveys Tutorials* 17, 4 (Fourthquarter 2015), 2347–2376.

[2] APIs.guru. [n.d.]. API tooling for better developer experience. https://apis.guru/.

[3] AsyncAPI Initiative. [n.d.]. AsyncAPI specification 2.0.0. URL: https://www.asyncapi.com/docs/specifications/2.0.0/, last accessed May 2020.

[4] Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz-Cortés, and Mike Hinchey. 2014. An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. *Journal of Systems and Software* 91 (2014), 3 – 23.

[5] Federico Ciccozzi and Romina Spalazzese. 2017. MDE4IoT: Supporting the Internet of Things with Model-Driven Engineering. In *Intelligent Distributed Computing X*, Costin Badica, Amal El Fallah Seghrouchni, Aurélie Beynier, David Camacho, Cédric Herpson, Koen Hindriks, and Paulo Novais (Eds.). Springer International Publishing, Cham, 67–76.

[6] Tony Clark and Balbir S. Barn. 2012. A Common Basis for Modelling Service-Oriented and Event-Driven Architecture. In *Proceedings of the 5th India Software*

[9] https://megamart2-ecsel.eu

[10] A thorough evaluation of development time reduction is still in progress.

Engineering Conference (Kanpur, India) *(ISEC '12)*. Association for Computing Machinery, New York, NY, USA, 23–32. https://doi.org/10.1145/2134254.2134258

[7] Tuhin Kanti Das and Juergen Dingel. 2018. Model development guidelines for UML-RT: conventions, patterns and antipatterns. *Software and Systems Modeling* 17, 3 (2018), 717–752. https://doi.org/10.1007/s10270-016-0549-6

[8] Flávia C. Delicato, Paulo F. Pires, and Thais Batista. 2013. *Middleware Solutions for the Internet of Things*. Springer Publishing Company, Incorporated, London.

[9] Oscar Díaz and Felipe M. Villoria. 2010. Generating blogs out of product catalogues: An MDE approach. *J. Syst. Softw.* 83, 10 (2010), 1970–1982. https://doi.org/10.1016/j.jss.2010.05.075

[10] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2018. Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach. In *22nd IEEE International Enterprise Distributed Object Computing Conference, EDOC 2018, Stockholm, Sweden, October 16-19, 2018*. 181–190. https://doi.org/10.1109/EDOC.2018.00031

[11] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2018. OpenAPItoUML: A Tool to Generate UML Models from OpenAPI Definitions. In *Web Engineering - 18th International Conference, ICWE 2018, Cáceres, Spain, June 5-8, 2018, Proceedings*. 487–491. https://doi.org/10.1007/978-3-319-91662-0_41

[12] Dave Evans. 2011. The internet of things: How the next evolution of the internet is changing everything. *CISCO white paper* 1, 2011 (2011), 1–11.

[13] Roy Thomas Fielding. 2000. *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation. University of California, Irvine.

[14] The Eclipse Foundation. [n.d.]. Eclipse Modeling Project - Eclipse Modeling Framework - Home. http://www.eclipse.org/emf/, last accessed May 2020.

[15] The Eclipse Foundation. [n.d.]. Xtext - Language Engineering Made Easy! http://www.eclipse.org/Xtext/, last accessed May 2020.

[16] Martin Fowler. 2010. *Domain Specific Languages* (1st ed.). Addison-Wesley Professional.

[17] Aitziber Iglesias, Markel Iglesias-Urkia, Beatriz López-Davalillo, Santiago Charramendieta, and Aitor Urbieta. 2019. TRILATERAL: Software Product Line based Multidomain IoT Artifact Generation for Industrial CPS. In *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,*. INSTICC, SciTePress, 64–73. https://doi.org/10.5220/0007343500640073

[18] Markel Iglesias-Urkia, Abel Gómez, Diego Casado-Mansilla, and Aitor Urbieta. 2020. Automatic generation of Web of Things servients using Thing Descriptions. *Personal and Ubiquitous Computing* (July 2020). https://doi.org/10.1007/s00779-020-01413-3

[19] Markel Iglesias-Urkia, Abel Gómez, Diego Casado-Mansilla, and Aitor Urbieta. 2019. Enabling easy web of things compatible device generation using a model-driven engineering approach. *ACM International Conference Proceeding Series* (2019). https://doi.org/10.1145/3365871.3365898

[20] Markel Iglesias-Urkia, Aitziber Iglesias, Beatriz López-Davalillo, Santiago Charramendieta, Diego Casado-Mansilla, Goiuria Sagardui, and Aitor Urbieta. 2020. TRILATERAL: A Model-Based Approach for Industrial CPS – Monitoring and Control. In *Model-Driven Engineering and Software Development*, Slimane Hammoudi, Luís Ferreira Pires, and Bran Selić (Eds.). Springer International Publishing, Cham, 376–398.

[21] Ana Ivanchikj and Cesare Pautasso. 2020. *Modeling Microservice Conversations with RESTalk*. Springer International Publishing, Cham, 129–146. https://doi.org/10.1007/978-3-030-31646-4_6

[22] Nasser Jazdi. 2014. Cyber physical systems in the context of Industry 4.0. In *2014 IEEE International Conference on Automation, Quality and Testing, Robotics*. 1–4. https://doi.org/10.1109/AQTR.2014.6857843

[23] Henning Kagermann, Johannes Helbig, Ariane Hellinger, and Wolfgang Wahlster. 2013. *Recommendations for implementing the strategic initiative INDUSTRIE 4.0: Securing the future of German manufacturing industry; final report of the Industrie 4.0 Working Group*. Forschungsunion.

[24] Stefan Klikovits, Alban Linard, and Didier Buchs. 2018. CREST - A DSL for Reactive Cyber-Physical Systems. In *System Analysis and Modeling. Languages, Methods, and Tools for Systems Engineering*, Ferhat Khendek and Reinhard Gotzhein (Eds.). Springer International Publishing, Cham, 29–45.

[25] Paulo Leitão, Armando Walter Colombo, and Stamatis Karnouskos. 2016. Industrial automation based on cyber-physical systems technologies: Prototype implementations and challenges. *Computers in Industry* 81 (2016), 11–25.

[26] Brenda M Michelson. 2006. Event-driven architecture overview. *Patricia Seybold Group* 2, 12 (2006), 10–1571.

[27] Behailu Negash, Tomi Westerlund, Amir M. Rahmani, Pasi Liljeberg, and Hannu Tenhunen. 2017. DoS-IL: A Domain Specific Internet of Things Language for Resource Constrained Devices. In *Procedia Computer Science*, Shakshuki E. (Ed.), Vol. 109. Elsevier B.V., 416–423. https://doi.org/10.1016/j.procs.2017.05.411

[28] OMG. [n.d.]. Meta Object Facility (MOF), Ver. 2.5.1. http://www.omg.org/spec/MOF/2.5.1/.

[29] OpenAPI Initiative. [n.d.]. OpenAPI Specification. URL: https://github.com/OAI/OpenAPI-Specification, last accessed May 2020.

[30] Till Riedel, Nicolaie Fantana, Adrian Genaid, Dimitar Yordanov, Hedda R. Schmidtke, and Michael Beigl. 2010. Using web service gateways and code generation for sustainable IoT system development. In *2010 Internet of Things (IOT)*. Tokyo, Japan, 1–8. https://doi.org/10.1109/IOT.2010.5678449

[31] Klaus Schwab. 2017. *The Fourth Industrial Revolution*. Crown Publishing Group.

[32] Manfred Sneps-Sneppe and Dmitry Namiot. 2016. On web-based domain-specific language for Internet of Things. In *International Congress on Ultra Modern Telecommunications and Control Systems and Workshops*, Vol. 2016-January. IEEE Computer Society, 287–292. https://doi.org/10.1109/ICUMT.2015.7382444

[33] SmartBear Software. [n.d.]. What Is OpenAPI? https://swagger.io/docs/specification/about/.

[34] Yahya Tashtoush, Mohammed Nour AlRashdan, Osama Salameh, and Mohamamd Alsmirat. 2019. Swagger-based jQuery Ajax Validation. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*. 0069–0072. https://doi.org/10.1109/CCWC.2019.8666542

[35] Kleanthis Thramboulidis and Foivos Christoulakis. 2016. UML4IoT—A UML-based approach to exploit IoT in cyber-physical manufacturing systems. *Computers in Industry* 82 (2016), 259–272. https://doi.org/10.1016/j.compind.2016.05.010

[36] W3C. 2019. Web of Things at W3C. https://www.w3.org/WoT/.

[37] Bobbi Young, Judd Cheatwood, Todd Peterson, Rick Flores, and Paul C. Clements. 2017. Product Line Engineering Meets Model Based Engineering in the Defense and Automotive Industries. In *Proceedings of the 21st International Systems and Software Product Line Conference, SPLC 2017, Volume A*. 175–179.